Angelaz-mturek-nedmonds

Revised design document

Purpose:

Design a collaborative editor that allows multiple users to work on a single document simultaneously across a network.

Introduction

We initially planned to use a double-linked-list-based representation for the document with each of the nodes containing an element of the text (the size of which could depend on the implementation but would most likely be a single character). Each of the nodes would have been identified by its unique ID which would prevent any confusion about indices as other users' changes modify the text. This way, we could achieve true concurrency as multiple nodes could be modified at the exact same time without interfering with each other. Minor issues related to this system (such as finding the ID's of the elements changed in the GUI where we only see a String representation of the linked-list) would have been possible to resolve and we drafted some of the proposals along with the entire initial design in Appendix B.

As the number of issues related to this more robust but much more complex implementations kept increasing (the number of messages a server processing the necessary requests grew, desired behavior required the addition of many ad hoc work-arounds etc.), we decided to recreate the initial design concentrating on simplicity as opposed to a design that's elegant as an idea but more complicated in implementation.

The new design removes true concurrency and implements a system of 2 queues, one held by the server and one by each of the users. The main points of this design are:

1. The changes made by a user in their GUI are immediately reflected in what they see but their local copy of the document is not directly mutated.
2. The local copy held by a user is just a simple String, not a linked list
3. Only the server has the capability to mutate individual users' local copies of the document.
4. The server keeps a shared queue of all the changes made by all the users in the order they are received. Along with point 3, this guarantees that at the end of the day, all users must necessarily hold the same local copy of the document, regardless of the order of changes they see in their GUI.
5. The actual content of the document is never directly distributed and the server does not directly store it. Each document is represented as a series of changes (i.e. differences from the previous version) that when applied in order can recreate the document at any state since it's creation.

Design document

Server:

The role of the server is very limited in this design since it does not directly store a copy of the document nor does it apply to it the changes received from individual users. Main point of the server is to hold an instance of an ArrayList called ***serverHistory*** that accumulates all the changes applied to the document since version 0 (which is by default an empty document). The position of a **ServerRequestDQ** (the grammar of which will be specified later) in the array list is the number of the version that applying that particular request will produce (it is also attached to the message as its last segment).

When a client connects to the server, the server assigns him a unique user ID in a hello message and then starts listening to incoming requests from the client. Once a change request is received from the user, the server needs to make sure that the indices of that change match the latest version handled by the server. Since we know which version the client was holding when he made that change and we know that all the local changes made by the client in his ***localQueue*** prior to this one must have been already been processed by the server, we simply update the current request by applying all the requests in serverHistory other users made since the version held by the client. A concrete example:

- Two users both hold a document in version 5 with content ABCD
- User 1 inserts an X after the A → "INSERT|1|0|X" with version ID 5
- User 2 inserts a Y after the C → "INSERT|3|0|Y" with version ID 5
- The request sent by User 1 gets processed first and it is just included as it is since the version ID matches the version held by the server
- The request sent by User 2 gets processed second and the server sees that since User 2 sent it not knowing User 1 inserted an X in the text. In order to synchronize the insertion index with the current version (6), the INSERT|1|0|X request is ***"applied"*** to the "INSERT|3|0|Y request, updating it to "INSERT|**4**|0|Y. The updated request is then added to serverHistory and distributed to all the connected clients.

Client:

The client consists of two parts, the actual GUI which groups the view and the controller and the UserDQ class which manages the local version of the document and thus represents the model.

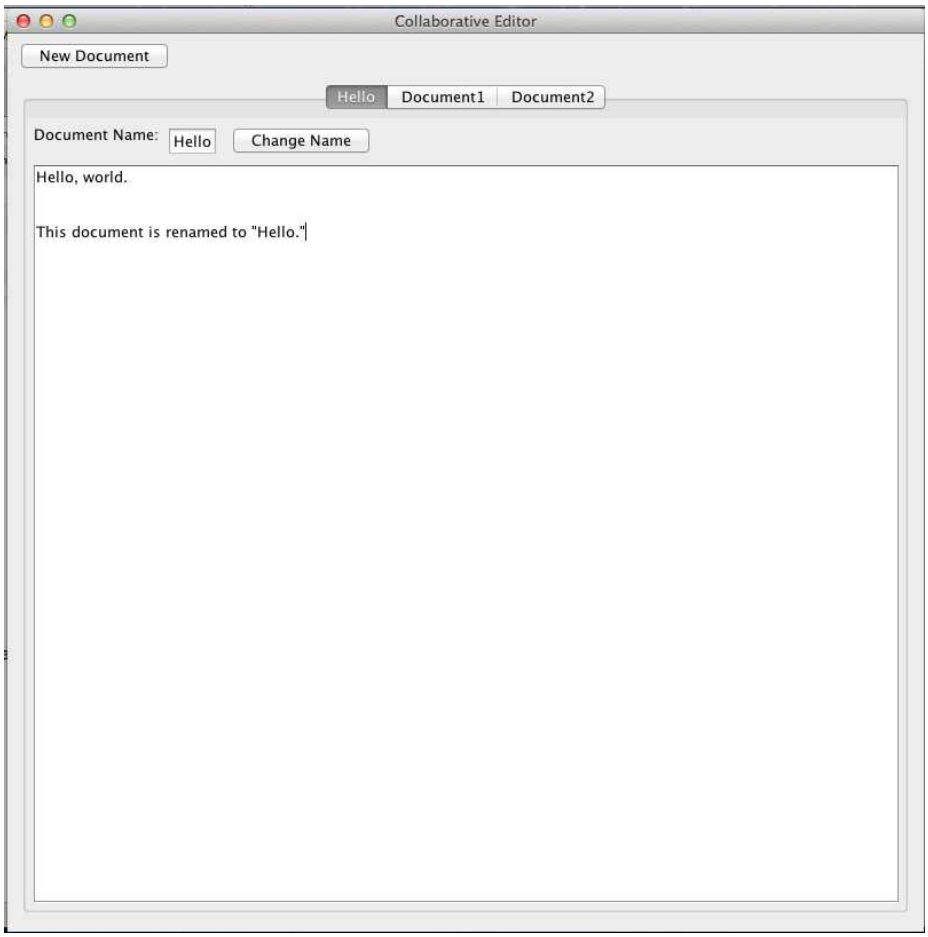
The EditorGUI class manages the connection of a user to the server and displays the current version of the document provided by the model. The GUI responds to 2 kinds of events:

1. The user directly modifies the content of his JTextPane. We do not want this to affect the local version of the document, however, since this would result in data inconsistencies over time and a lot of effort to ensure proper synchronization. Instead, the model places the change into its own ***localQueue*** which is an ArrayList containing a sequence of changes to be applied to the latest synchronized copy held by the client. The synchronized local copy (***syncCopy***) may only be modified by the server, never by a direct change in the user interface. The client then sends this change to the server to be processed and distributed to other users.
2. A change request is received from the server in a background thread, if this change is coming from this user, it gets removed from the localQueue (as it has been processed and is part of the current syncCopy). If the change is coming from another user, it is applied to the syncCopy (thus increasing its version number) and also to all the requests in localQueue (since those are only meaningful relative to the syncCopy version which has changes).

Once all the events have been processed, the GUI updates what the user sees by asking the model for the current **viewCopy –** UserDQ then takes the syncCopy and applies all the changes stacked in

localQueue to get what the user is supposed to be seeing.

The GUI itself will be built with the Java Swing library. It will contain: 1) a “New Document” button that creates a new document, 2) a “Change Name” button that renames the document, 3) several tabs, each containing a document, and 4) the text pane that contains the body of the document.



Messages:

There are several kinds of messages that get passed between the server and the client at different points in the execution of the program. Following a successful connection to the server, a client receives the ***hello*** message that assigns him his User ID and will eventually give him the list of all the documents available on the server. The client also requests all the changes made to the document it is opening to get to the current server version. This is done through the ***load*** message.

During most of the time, all communication between the client and server will be done through ***ServerRequestDQ*** messages that specify changes to be applied to the document → a client sends one when a changes is made in a GUI and the server distributes them to all the other clients.

At the end of the communication, a client sends a ***bye*** message which disconnects all the open sockets and stream handlers.

Hello and bye messages:

Maintenance messages:

```
Hello ::= HELLO|UserID
Bye :: BYE
Load ::= LOAD|DocumentID
```

ServerRequestDQ:

Unlike in our previous design, we decided that for simplicity all changes should be applied as block changes (since single-character insertion and deletion are just block edits of length 1) and that the proposed system of anchors is not necessary in a String-representation based design. Therefore, we were able to decrease the number of request-related messages just to ***insert*** and ***delete***.

Each of the requests sent between the client and the server begins with the ***user ID*** of the user who placed the request and a ***request number*** which is used for the identification of a request when a user receives it. Each user assigns a request number to each of his requests and increases the request counter after every one, so that each request number is unique for a given user.

The main part of the message (the ***body*** that directly depends on an input by the user in his GUI) contains the ***identification number of the document*** being edited, the ***type of the action*** applied, its ***beginning***, ***end***, and the ***content*** of the request. Obviously, the value of end can be arbitrary for inserts, whereas the content is irrelevant for deletes. Example bodies:

- 0|INSERT|3|123|X → insert X at index 3 in document 0
  - Produces the change: ABCDE → ABCXDE
- 0|DELETE|1|3|XXX → delete segment between index 1 (inclusive) and 3 (exclusive)
  - Produces the change: ABCDE → ADE

Each request closes with a ***version ID***, which has a different meaning for messages coming from the server to the client and in the opposite direction.

- For messages sent from the client to the server, the version ID is the number of the last version the client downloaded from the server → it's the version of his syncCopy.
- For messages sent from the server to the client, the version ID is simply the index of that particular change in ***serverHistory*** and therefore also the version number of the local copy the client will hold after processing the message.

Careful versioning is necessary to preserve the consistency of edits between clients.
The complete grammar for a String representation of a ServerRequestDQ (which is completely interchangeable with an actual ServerRequestDQ by the use of the built-in constructor and the toString method:

```
ServerRequestDQ ::= Head|Body|Tail
Head ::= UserID|RequestNumber
Body ::= DocumentID|Action|Beginning|End|Content
Tail ::= VersionID

UserID ::= a String not including "|"
RequestNumber ::= non-negative int
DocumentID ::= non-negative int
Action ::= INSERT|DELETE
Beginning ::= non-negative int
End ::= non-negative int
Content ::= a String non-including "|"
VersionID ::= non-negative int
```

overall: USER_ID|REQ_NUM|DOC_ID|ACTION|BEG|END|CONTENT|VERSION_ID

Appendix A: Diagrams: